Dataflow-Driven GPU Performance Projection for Multi-Kernel Transformations

Jiayuan Meng, Vitali A. Morozov, Venkatram Vishwanath, Kalyan Kumaran
Argonne National Laboratory
{jmeng, morozov, venkatv, kumaran}@alcf.anl.gov

***Abstract*—Applications often have a sequence of parallel operations to be offloaded to graphics processors; each operation can become an individual GPU kernel. Developers typically explore a variety of transformations for each kernel. Furthermore, it is well known that efficient data management is critical in achieving high GPU performance and that "fusing" multiple kernels into one may greatly improve data locality. Doing so, however, requires transformations *across* multiple, potentially nested, parallel loops; at the same time, the original code semantics and data dependency must be preserved. Since each kernel may have distinct data access patterns, their combined dataflow can be nontrivial. As a result, the complexity of multi-kernel transformations often leads to significant effort with no guarantee of performance benefits.**

This paper proposes a dataflow-driven analytical framework to project GPU performance for a sequence of parallel operations. Users need only provide CPU code skeletons for a sequence of parallel loops. The framework can then automatically identify opportunities for multi-kernel transformations and data management. It is also able to project the overall performance *without* implementing GPU code or using physical hardware.

I. Introduction

Accelerators such as graphics processors (GPUs) are increasingly being adopted in high performance computing. They are particularly attractive for data-parallel, compute-intensive workloads. Many scientific applications, however, are data-intensive workloads and often have poor GPU performance due to the limited L1 storage per thread and bandwidth. Thus, developers have to exhaustively explore code transformations to reuse and shared data as much as possible. These transformations may not be limited within boundaries of individual parallel loops. For applications that have a sequence of parallel operations, a common transformation is to fuse parallel operations into one in order to improve data locality. This is also referred to as *kernel fusion* in GPU optimizations [42]; data can be cached in the L1 storage (e.g., shared memory[1]) and reused within the fused kernel rather than stored into the device memory and reloaded in a separate kernel. In order to perform multi-kernel transformations, three questions need to be answered:

1) **Which kernels to fuse and in what order?** Knowledge about the dataflow among multiple kernels is needed. Kernels touching different data sections do not benefit from fusion. Dependent kernels have to be fused in a way that the order of dependent operations is preserved.
2) **How to fuse parallel `for` loops (which may be shaped differently)?** Each kernel may have a different *loop space* that includes multiple loop iterators each having a unique range. The kernels to fuse will eventually share a common set of parallel loop iterators. In the original code, however, these kernels may not have loops with the same number of iterations or dimensionality. In addition, the loop iterators may have a more complex mapping than a direct correspondence.
3) **How to optimize data movement for the fused kernel (which may have multiple, nested loops)?** Data elements can be reused or shared in new ways in the fused kernel. Issues to investigate include which arrays or array segments to cache, when to insert the explicit caching instructions, and how to effectively utilize L1 storage. Note that data reuse and efficient caching strategies may not be obvious if data accesses are buried deep within separated nested loops. We refer to dataflows involving multiple nested loops as *hierarchical dataflows*. To our knowledge, current techniques, even those targeting CPU code optimizations, do not address hierarchical dataflows.

Addressing these questions requires a comprehensive understanding of the individual kernels and the interactions among the multiple kernels. There are several tradeoff factors, including computation intensity, amount of global memory accesses, usage of L1 storage space, degree of latency hiding, and the ability to coalesce SIMD memory accesses. Without careful tuning, these side effects may negatively impact performance. Conventionally, evaluating these tradeoffs requires implementing and running many versions of GPU codes on the physical hardware. This process can be time consuming, expensive, and sometimes even intractable. Since tuning a single GPU kernel is already a challenging task, multi-kernel transformation may easily fall off a developer's agenda, not to mention the risk of low or even negative performance benefit.

We propose a framework to transform and project the performance of *a sequence* of parallel operations on GPUs. We restrict our focus to the scenario where *data is already loaded in GPU memory*. Determining which part of the application to be loaded onto the GPU and balancing between CPU and GPU workloads are complementary work and are beyond the scope of this paper. Our multi-kernel optimization focuses on data management and caching strategies, often the dominating factors for the resulting performance; instruction scheduling within the fused kernel remains future work. Our technical contributions include the following:

1) Demonstrate the use of code skeleton as an abstraction to model multiple kernels.
2) Propose an abstract intermediate representation for code skeletons to ease high-level transformations and dataflow analysis from the perspective of memory footprints of code blocks rather than instruction operands.
3) Identify regional dependencies among kernels using memory footprint analysis.
4) Establish a dataflow-based algorithm to derive a shared loop space for multiple parallel loops.
5) Explore caching strategies of the fused kernel with hierarchical dataflow analysis and transformations.
6) Devise a dynamic programming algorithm to efficiently explore the multi-kernel transformation space.

These techniques are not GPU-specific; they have the potential to be integrated into several performance tools. In this paper, we demonstrate the techniques by integrating them into

[1]This paper focuses on explicit caching using GPUs' shared memory, which is often the primary option of cache optimization even for state-of-the-art GPUs with implicit cache hierarchies. We leave the study of projecting GPU transformations for implicit caching as future work.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

GROPHECY [29], a GPU performance projection tool originally built for individual kernels. We tested the framework with four benchmarks involving a total of eight GPU kernels. Projections are compared with manual optimizations over two GPU generations. The projection error recorded during our experiments averages at 14%. Our framework is able to identify multi-kernel transformations that match manual optimizations, and are successfully able to project whether the efforts are worthwhile *without needed to either implement GPU code or acquire the GPU hardware*.

The remainder of this paper is organized as follows. Section II presents the overall multi-kernel GPU projection framework. Section III describes the benchmarks used in our case studies. Section IV introduces basic terminology and operations used in our analysis and transformations. Section V demonstrates how to identify and fuse multiple kernels. Section VI proposes hierarchical dataflow analysis and transformation for cache optimization. Section VII showcases the use of the framework from different perspectives. Section VIII discusses literature related to our work. Conclusions are drawn in Section IX.

II. Overall Framework

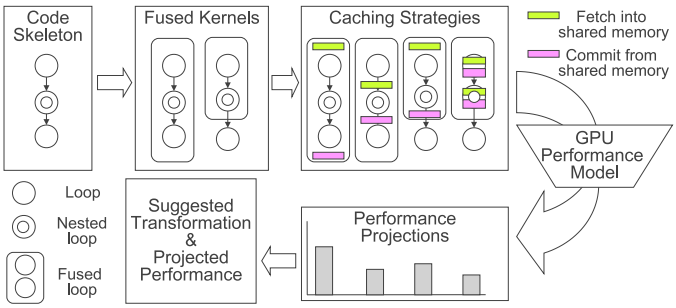


Fig. 1: Overview of the framework.

Figure 1 summarizes the major steps in our framework. To prepare the input, users need to abstract a sequence of parallel operations as a CPU *code skeleton* (see Section II-A). We suggest users preserve individual operations in the code skeleton instead of fusing them. This is typically the way that users develop the initial code—fusing operations often occur in later development stages for optimization purposes. In this way, each operation exposes its parallelism to the maximum, opening room for more optimization opportunities. Our technique is then able to determine which kernels can be fused and how to fit multiple kernels into a common loop space. For each possible fused kernel, the framework investigates the merged dataflow and explores cache optimizations at each level of nested loops. This step generates a number of transformed code skeletons. Using our prior techniques in synthesizing performance characteristics and adopting the GPU performance model [18], [29], the framework projects the performance for each transformed code skeleton. Eventually, it reports the best projected performance together with the corresponding code skeleton so that users can implement GPU code accordingly.

The framework is modular, and allows our multi-kernel analyses and transformation techniques to be applied separately in other scenarios. Although the framework into which we integrate our techniques requires code skeletons from users, the actual input to our technique is the block skeleton tree (see Section IV-A), which may be derived automatically from static analyses in the future. Moreover, the framework can employ future hardware performance models to project performance for current and future GPU architectures.

A. Code Skeletons

The code skeleton is a high level language to summarize structures and performance properties of an application in a concise and intuitive form [29]. The performance structure can include control flow, data flow, and computation intensity. It encourages users to express high level knowledge such as nested parallel `for` loops (`forall`) and streaming loops (`stream`) in which consumed data will be no longer used in subsequent iterations.

For conditional branches, the code skeleton either retains the dominant control flow path if a single such path exists, else, concatenates both branch paths to mimic a GPU's SIMD branch handling mechanism wherein latencies for both divergent branch paths are charged to each thread. A more comprehensive technique to handle branches in the code skeleton will be studied in the future.

For illustration purposes, we use code skeletons to demonstrate the complexity of input CPU kernels in Section III, and the diversity of output code transformations in Section VII. Because of space limitations, the presented code skeletons show only loop structures and data access patterns. Statistics about computation instructions are presented in our prior work [29].

III. Benchmarks for Case Studies

We present four multi-kernel benchmarks that are key components in representative applications in the areas of medical imaging, microprocessor design, fluid dynamics, and quantum physics. `SRAD`, `HotSpot`, and `CFD` are multi-kernel benchmarks found in the Rodinia benchmark suite [9]. `Stassuij` is extracted from a DOE INCITE application. The key properties of the benchmarks are summarized in Table I.

`SRAD` is a diffusion method to remove speckles for ultrasonic and radar imaging applications without destroying important image features. It has two kernels: the first one generates diffusion coefficients, and the second one updates the image. Data dependency among the two kernels involves several arrays, and each data-parallel task in the consumer kernel depends on several tasks in the producer kernel. The code structure is shown in Listing 1 in the form of code skeletons

`HotSpot` is an ordinary differential equation solver over a structured grid to estimate micro-architecture temperature. Every element is computed by gathering a 3×3 neighborhood of elements (i.e., the stencil) from the input array. Listing 2 depicts a single kernel repeated iteratively to simulate time-steps. In fact, several invocations of the same kernel across several iterations can be fused together.

`CFD` is an unstructured-grid, finite-volume solver for the 3-D Euler equations for compressible flow. The core part of the benchmark is spread over three GPU kernels. The code structure is illustrated in the form of code skeletons in Listing 3. The two kernels are separated in order to enforce global synchronization so that the array `variables` can be consumed before it is updated. By exploiting double buffering, this global synchronization can eliminated, and the two kernels can be merged.

`Stassuij` lies in the core of Green's Function Monte Carlo [20], [33], [34], which performs Monte Carlo calculation for light nuclei. It multiplies a 132×132 sparse matrix of real numbers with a 132×2048 dense matrix of complex numbers. The sparse matrix `A` is compressed and represented with three vectors: `T`, `J`, and `I`. `T[n]` stores the values of nonzero elements. `A`'s first nonzero element in the jth row corresponds to the `T` element with the index of `J[j]`. `I[n]` stores the column number of the nth element in T. In other words, the mth nonzero element in the jth row of `A` is $A[j][I[J[j]+m]]$,

Listing 1: Separate SRAD kernels.

```
R = 4096, C = 4096
float a[R][C], b[R][C]
float c[R][C], d[4][R][C]
forall i=0:R,j=0:C
{
  read a[i-1][j]
  read a[i+1][j]
  read a[i][j-1]
  read a[i][j+1]
  for k=0:4
    write d[k][i][j]
  write c[i][j]
}
forall i=0:R,j=0:C
{
  read c[i+1][j]
  read c[i][j+1]
  read a[i][j]
  for k=0:4
    read d[k][i][j]
  write b[i][j]
}
```

Listing 2: The self-repeating HotSpot kernel.

```
R = 1024, C = 1024, N = 120
float a[R][C], b[R][C], pow[R][C]
/* double buffering */
src=a; dst=b
/* Iterate over a kernel */
for n=0:N
{
  forall r=0:R,c=0:C
  {
    /* Need data from several
     * producer iterations
     */
    read src[r][c]
    read src[r][c+1]
    read src[r][c-1]
    read src[r-1][c]
    read src[r+1][c]
    read pow[r][c]
    write dst[r][c]
  }
  swap=src; src=dst; dst=swap
}
```

Listing 3: Separate CFD kernels, with the last kernel depending on the previous two.

```
NNB = 4, NVAR = 5
NELR = 193280
float src[NELR*NVAR]
float dst[NELR*NVAR]
float step_factors[NELR]
int neighbors[NELR*NNB]
float fluxes[NELR*NVAR]

forall i=0:NELR
{
  for k=1:4
    /* Compound accesses w/t
     * multiple loop indices
     */
    read src[i+k*NELR]
  write step_factors[i]
}

forall i=0:NELR
{
  for j=0:NNB
  {
    /* Indirect accesses */
    nb = neighbors[i + j*NELR]
    for k=0:NVAR
      read src[nb + k*NELR]
  }
  for k=0:NVAR
    write fluxes[i + k*NELR]
}

/* Depend on two producer loops */
forall i = 0:NELR
{
  read step_factors[i]
  for k = 0:NVAR
  {
    read fluxes[i + k*NELR]
    write dst[i + k*NELR]
  }
}
```

Listing 4: Separate Stassuij kernels with differently shaped loop spaces.

```
NT = 132
NS = 2048
NSG = 512
ELEMS = 1848
avg_j_ntdt = 14
int J[NT+1], I[ELEMS], T[ELEMS]
float A[NT][NS][2]
float B[NT][NS][2]
float C[NT][NS][2]

forall j=0:NT, i=0:NS, r=0:2
{
  read J[j]
  read J[j+1]
  stream n = 0:avg_j_ntdt
  {
    read T[n]
    read I[n]
    read A[I[n]][i][r]
  }
  write B[j][i][r]
}

/* Loop with a different shape */
forall j=0:NT, ig=0:NSG, m=0:4, r=0:2
{
    read M[ig][m]
    /* Indirect accesses */
    read A[j][M[ig][m]][r]
    read B[j][M[ig][m]][r]
    write C[j][M[ig][m]][r]
}
```

TABLE I: Workload properties

Benchmark	Key Properties	Language	No. of Kernels	Input Size
SRAD	Structured grid. Data dependency involves multiple arrays; each points to different producer iterations	C++	2	4096×4096
HotSpot	Structured grid. Iterative, self-dependent kernels. A deep dependency chain among dependent kernels	C++	1-4 (Repeated)	1024×1024
CFD	Unstructured grid. Indirect, compound array accesses. Nested loops. One-to-one correspondence between consumer and producer iterations The consumer loop has multiple producer loops	C++	3	193280 elements each with 4 neighbors
Stassuij	Sparse linear algebra and spectral methods. Nested loops. Dependent parallel loops with different shapes. Dependency involves indirectly accessed arrays	Fortran	2	$T[132][132] \times A[132][2048]$, where T has 1848 nonzero elements

which corresponds to $T[J[j] + m]$. Listing 4 summarizes its structure.

The benchmarks demonstrate different opportunities and challenges in multi-kernel transformations. SRAD (Listing 1) exemplifies a simple multi-loop code skeleton. The loop space of parallel operations need not have the same shape nor the same size, as Stassuij (Listing 4) shows. Moreover, a parallel operation can depend on several previous operations; these operations can access the same array with different patterns. Data can be accessed indirectly, as is the case for CFD (Listing 3). Furthermore, forall loops can be repeatedly executed, leading to a circular dependency as illustrated in Listing 2.

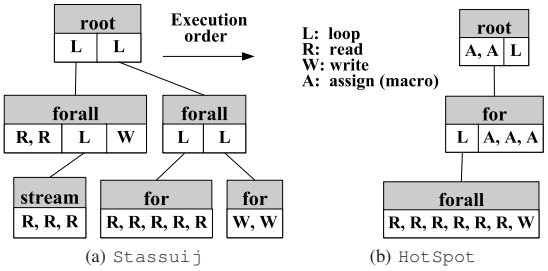
IV. Basic Terms and Representations

In this section, we introduce the basic terms and data structures used in the paper. They play a key role in analyzing and transforming multi-kernel codes.

A. Block Skeleton Tree (BST)

A code skeleton is parsed into an intermediate representation named the *block skeleton tree* (BST). Each node of the BST corresponds to a perfectly nested loop statement in the code skeleton. Imperfectly nested outer and inner loops become parent and child nodes, with leaf nodes representing innermost loops. Each node records the loop iterators and their ranges. It also records the loop body as a sequence of code skeleton statements. Compared with the abstract syntax tree (AST) in compilers whose leaf nodes can be individual operators and operands, the BST representation works at a much higher level. It encapsulates knowledge about flexibilities in the code structure and performance-related behaviors. For example, a node corresponding to a parallel loop can be partitioned; perfectly nested outer and inner loops can be interchangeable; array accesses can be reordered or moved to outer loops as long as dependency is preserved. Figure 2a and Figure 2b demonstrate

the BSTs for `Stassuij` and `HotSpot`, whose code skeletons are listed in Listing 4 and Listing 2, respectively.



(a) `Stassuij` (b) `HotSpot`

Fig. 2: Examples of block skeleton trees.

B. Tiles, Patterns, and Loops

When executed in a loop, an array access statement touches multiple elements. We use bounded regular section (BRS) [16] to study regional array accesses. In BRS, an array access can be regarded as a function that maps a range of loop iterators to a set of elements over the array. In this paper, we refer to the range of loop iterators as a *tile* ($\mathbb{T}$) and the set of accessed array elements as a *pattern* ($\mathbb{P}$). For example, suppose A is a 2-D array and an instruction accesses $A[r][c]$ in a nested `for` loop with two iterators, r and c. The tile corresponding to the loop is denoted $\mathbb{T}(r, c) = \{r : \langle r^l : r^u : r^s \rangle; c : \langle c^l : c^u : c^s \rangle\}$, where each of the three components represents the lower bound, upper bound, and stride, respectively. The overall pattern accessed within this loop is denoted $A[\langle r^l : r^u : r^s \rangle][\langle c^l : c^u : c^s \rangle]$, which is summarized by $\mathbb{P}(A[r][c], \mathbb{T}(r, c))$.

We have extended BRS so that indices of an array can be a function involving multiple loop iterators (e.g., $\mathbb{P}(A[foo(J|J \subseteq I)], \mathbb{T}(I = \{i_0, i_1, ...\}))$, where A is a 1-D array). Moreover, array references can be indirect accesses (e.g., $\mathbb{P}(A[foo(C[i])], \mathbb{T}(i))$, where C is an array of indices to array A).

BRS also establishes common pattern operators including `count`, `union`, and `intersect`. The operator `count`($\mathbb{P}$) returns the number of data elements in a pattern. Given a tile $\mathbb{T}(I)$ where $I = \{i_0, i_1, ...\}$ is a set of iterators, and an access statement $A[foo(J)][bar(K)]$ with $J, K \subseteq I$, we can calculate $count(\mathbb{P}(A[foo(J)][bar(K)], \mathbb{T}(I)))$ as $\prod_{i \in (J \bigcup K)} \frac{i^u - i^l}{i^s}$. Multiple patterns are merged by `union`($\mathbb{P}_0 \bigcup \mathbb{P}_1$) into one pattern that is the minimum BRS that includes all merged patterns. The operator `intersect`($\mathbb{P}_0 \bigcap \mathbb{P}_1$) returns a pattern as the intersection of multiple patterns. Nonempty intersection indicates overlapping patterns.

A perfectly nested loop is denoted $\mathbb{L}(\mathbb{I}, \mathbb{B})$, where $\mathbb{I}$ is a tile containing the full ranges of all loop iterators and $\mathbb{B}$ is the loop body. $\mathbb{B} = [statements]$ contains a sequence of statements. The statements can include data accesses, computation statistics, and inner loops.

V. Multi-Kernel Analysis and Fusion

In this section, we show how to identify and fuse multiple dependent kernels. Section V-A describes how to detect dependency among kernels. Then, for each pair of dependent kernels, in each iteration of the consumer kernel, we calculate the iteration in the producer kernel which generates the requisite data. We discuss this in Section V-B. With such knowledge, our technique can generate the BST representing the fused thread block, as Section V-C describes.

A. Analyzing Inter-Kernel Dependency

As a first step in analyzing inter-kernel dependency, one needs to understand the overall data patterns read and written in each kernel. For this, we merge all overlapping patterns corresponding to all data accesses within a kernel. Read patterns and written patterns are merged separately.

Then, for each node of the BST, the child `forall` loops are scanned once in execution order to determine dependency. During the scan process, previously written patterns are merged to represent the data produced thus far. If a kernel reads in a pattern that overlaps with a previously written pattern, then dependency is established between the kernel and the most recent kernel that produced the written pattern. In our current implementation, both read-after-write and write-after-read dependencies are taken into account equally, therefore the ordering of fused loop bodies is enforced conservatively. A local dependency graph is constructed accordingly for each node in the BST that has child `forall` loops.

The dependency graph can be partitioned in various ways; each partition can be fused into one kernel. By default, the framework uses a greedy scheme and fuses the longest dependency chain, unless users provide hints about *which* kernels to fuse. Note that users need not specify *how* kernels can be fused.

The loop dependency graph can be circular. For example, if a sequence of `forall` loops are executed repeatedly within an outer loop, one of the loops may produce data that is later consumed by the same loop or its producers in the next iteration. Common examples are iterative differential equation solvers such as `HotSpot` (Listing 2). In such cases, the outer loop is unrolled for a number of times so that the dependency chain can include multiple instances of the same set of loops, which can be fused later. Users can provide hints about how many times to unroll the outer loop.

B. Finding Dependent Producer Iterations

When fusing dependent `forall` loops, one needs to make sure an iteration of the consumer loop is executed *after* all dependent iterations of the producer loop. If the consumer loop is to be partitioned into parallel thread blocks, the same producer iteration may have to be replicated and inserted into several dependent thread blocks. Therefore, one needs to understand *which iterations* in the producer loop are needed for a given iteration in the consumer loop. This relationship is determined again by data access patterns.

Assume a producer kernel denoted by $L^p(\bar{i}^p, \mathbb{B}^p)$ and a consumer kernel denoted by $L^c(\bar{i}^c, \mathbb{B}^c)$. Assume that their dependency is caused by an N-D array A. Suppose an iteration in the producer writes to $A[f_0(\bar{i}^p)]...[f_N(\bar{i}^p)]$ and an iteration in the consumer reads from $A[g_0(\bar{i}^c)]...[g_N(\bar{i}^c)]$. The dependent producer iterations for a given consumer iteration can then be obtained by solving the equation set $G = F$ for $\bar{i}^p$ given $\bar{i}^c$, where $G = \{g_n | 0 \leq n \leq N\}$ and $F = \{f_n | 0 \leq n \leq N\}$.

This example is a simplification of the problem where only one pair of consumed and produced patterns are involved in dependency. In reality, several pairs of patterns can lead to dependency. As a more general solution for each pattern consumed by $\mathbb{L}^c$, we first identify the patterns produced by $\mathbb{L}^p$ that intersects with it. For each pair of dependent patterns, we solve $\bar{i}^p = F^{-1}(G(\bar{i}^c))$ to identify one dependent producer iteration. Eventually, we obtain the set of all dependent producer iterations for a given consumer iteration.

Note that this algorithm does not presume a one-to-one mapping between producer and consumer kernels' loop iterators. The mapping is based on dataflow and does not require producer

and consumer loops to have the same shape or follow the same data access patterns. In addition, a consumer iteration can depend on multiple producer iterations.

C. Constructing a Fused Thread Block

Knowledge about dependency at the granularity of loop iterations can be used to fuse a block of consumer and producer iterations to produce a fused thread block. The dependency mapping between consumer and producer iterations can be interpreted as a consumer iteration's "accesses" to the "array" representing the producer's loop space, expressed as $\mathbb{L}^p[\bar{i}^p] = \mathbb{L}^p[i_0^p][i_1^p]...[i_N^p]$. Given $\mathbb{T}^c$, which denotes the loop partition corresponding to a set of consumer iterations, the loop partition of the dependent producer iterations can be calculated as the "merged pattern" of all consumer iterations' accesses to the producer's loop space: $\bigcup_{0 \leq m \leq M}(\mathbb{P}(\mathbb{L}^p[\bar{i}_n^p], \mathbb{T}^c))$, where $\{\bar{i}_n^p | 0 \leq n \leq N\}$ refers to the set of dependent producer iterations of a given consumer iteration.

For example, by applying the algorithm described in Section V-B to `Stassuij` (Listing 4), we determine that an iteration of the second kernel, $\mathbb{L}^c[j][g][m][r]$, depends on an iteration in the first kernel, $\mathbb{L}^p[j][M[g][m]][r]$, where M is an array of indirect indices. By calculating the pattern of this "access" over a partition of the second `forall` loop, $T(j, g, m, r) = \{\langle 0:11:1\rangle; \langle 0:3:1\rangle; \langle 0:4:1\rangle; \langle 0:2:1\rangle\}$, we can obtain the dependent partition in the first `forall` loop as $T(j, i, r) = \{\langle 0:11:1\rangle; M[\langle 0:3:1\rangle][0:4]; \langle 0:2:1\rangle\}$.

In `HotSpot` (Listing 2), a consumer iteration $\mathbb{L}^c[r][c]$ depends on five producer iterations at $\mathbb{L}^p[r+1][c]$, $\mathbb{L}^p[r-1][c]$, $\mathbb{L}^p[r][c]$, $\mathbb{L}^p[r][c+1]$, and $\mathbb{L}^p[r][c-1]$. By calculating the patterns associated with all five "accesses" over a 16×16 consumer loop partition and merging the resulting patterns, we derive that a consumer loop partition, $T(r, c) = \{\langle 0:16:1\rangle; \langle 0:16:1\rangle\}$, depends on a producer loop partition at $T(r, c) = \{\langle -1:17:1\rangle; \langle -1:17:1\rangle\}$.

Because no particular execution order is enforced among parallel thread blocks, the fused kernel packs a partition of the consumer loop with all dependent producer iterations into a single thread block, *even if this means some producer iterations may be replicated in different thread blocks*. Note that within a thread block, the dependent producer partition can have a different size from that of the consumer partition. The thread block size is independent of any loop partitions and is constrained only by hardware limitations. If the loop partition contains more iterations than the thread block size, then some threads can compute multiple iterations. If the loop partition contains fewer iterations than the thread block size, the residue threads will stay idle when others are computing. Our technique explores the impact of different thread block sizes on each partitioning of the fused kernel.

The fused kernel for a producer `forall` loop, $\mathbb{L}^p$, and a consumer `forall` loop, $\mathbb{L}^c$, can be formally constructed as $\mathbb{L}^{fused} = \mathbb{L}(\bar{i}_{grid}^c, \mathbb{L}(\bar{i}_{block}^c, \mathbb{B}_{fused}))$, where $\bar{i}_{grid}^c$ and $\bar{i}_{block}^c$ are iterators over the grid of thread blocks and the threads within a thread block, respectively, according to the consumer's loop space. When a thread needs to execute one consumer iteration and N producer iterations, the fused loop body to be executed within a single thread is $\mathbb{B}_{fused} = [\mathbb{B}_0^p, \mathbb{B}_1^p, ..., \mathbb{B}_N^p, \mathbb{B}^c]$, where $\mathbb{B}_n^p$ equals $\mathbb{B}^p$ except that the iterators $\bar{i}^p$ are expressed as a function of $\bar{i}^c$. The process repeats until all loops in the target dependency chain are fused. Each fused kernel replaces those BST nodes corresponding to the original loops with a new subtree according to $\mathbb{L}^{fused}$.

VI. Hierarchical Caching

Fusing kernels has the potential to increase data reuse and cache efficiency. The key challenge in optimizing cache efficiency for a fused kernel is that the data is likely to be reused across nested loops. Hence, a data management heuristic to to explore inter-loop and multi-level cache optimizations is critical. We start by introducing a dataflow-based algorithm to explore caching strategies for a sequence of access statements at the same loop level. The algorithm is next extended for more complex cases involving multiple nested loops. We demonstrate the efficacy of the framework to efficiently search the transformation space, as well as various space exploration pruning techniques.

A. Caching for a Single Level of Loop

The goal is to explore ways to cache combinations of a sequence of data accesses in order to improve overall data reuse. A cached access statement exploits locality of data in shared memory. A constraint to consider here is the amount of shared memory space needed.

Given two access statements with overlapping patterns, if one of them is cached, it is often beneficial to cache the other and reuse data resident in the shared memory. Therefore, our framework first scans the sequence of accesses and puts those with overlapping patterns into *access groups*. Each access group is "closed under reuse"; for all iterations within a loop, an access reuses data only from another access in the same group, and the same data can be read and written several times within the same group of accesses.

The framework explores various caching strategies at this loop level, based on dataflow analysis *within* each access group. The algorithm goes through the following steps:

- Enumerate *which access statements to cache for.* By summing the sizes of all patterns in an entire access group and dividing the sum by the size of the merged pattern for the same group, we can calculate the average number of times data is reused within an access group. Similarly, the framework can obtain the degree of sharing among threads, as described in our prior work [29]. Access groups with reused or shared data become candidates for caching. Because of the limited capacity of the shared memory, not all reused or shared data can be cached. Our framework therefore enumerates all possible combinations of candidate access groups. For each combination, the set of access groups to cache is denoted G.
- Decide *what to load into the cache and when.* Data to load into the shared memory is obtained by merging overlapping patterns corresponding to the reads within each access group in G. A pattern should be loaded from the memory before the first read in its access group. The set of patterns to be loaded into the cache is denoted R.
- Decide *what to commit to the memory and when.* Data to be committed into the global memory is obtained by merging overlapping patterns corresponding to the writes within each access group in G. These patterns should be committed to the global memory at or after the last write in the sequence. The set of patterns to be committed is denoted W.
- Calculate *how much shared memory space to use and when to allocate and free the space.* For each access group in G, the required shared memory space equals the size of the merged pattern for all accesses in the group. The shared memory space should be allocated prior to the first access in the group and freed after the last access in the group.

Different access groups may reuse shared memory space if they have non-overlapping lifespan. We use an algorithm to scan the allocation and free events in the order of their occurrences to "emulate" shared memory usage trends. The maximum usage encountered during the scan is the overall shared memory space required for all accesses in G, denoted as S.

Thus, a caching strategy for a single level of loop can be summarized as $\mathbb{C} = \{G, R, W, S, T\}$, where T refers to *when* to insert data accesses to load and commit patterns in R and W to and from the shared memory. These instructions can be inserted in three ways:

- `In-place`: Augment the sequence of accesses with loads inserted right before the first read and commits inserted after the last write. Cached data need to accommodate only a single iteration.
- `In-parent`: The parent loop prepares and commits cached data used by all iterations of the loops.
- `In-stages`: Data is cached for every fixed number of iterations (i.e., a stage) in order to reduce shared memory footprint. Calculating staged patterns is discussed in our prior work [29] and is omitted in this paper because of space limitations.

Our framework explores all these possibilities. For `in-parent` placement, the corresponding data patterns that are left for parent loops to manage are denoted as $R^p \subseteq R$ and $W^p \subseteq W$, respectively. They themselves become additional, *brought-up accesses* in the parent loops. Data interactions between child and parent loops are further discussed below.

B. Caching for Multiple, Nested Loops

Computation code often exhibits a hierarchical loop structure; the BST can be an arbitrary tree in which a node can have multiple siblings and child nodes. The above dataflow analysis can be applied to leaf nodes (i.e., innermost loops) in a BST; for an inner node (i.e., an outer loop), in addition to its native data accesses, brought-up accesses can come from multiple child nodes. These brought-up accesses interact with native accesses, affecting how data is reused and shared, as well as how much shared memory space is required at the parent node. As a result, optimizing such hierarchical dataflow needs analysis not only between outer and inner loops but also among sibling loops. Different combinations of child nodes' caching strategies lead to different brought-up accesses and therefore different caching strategies in the parent node.

As Figure 3 illustrates, given a BST node, $\mathbb{L}$, with N child nodes and a combination of child nodes' caching strategies as $\{\mathbb{C}_i^c | 0 \leq i \leq N\}$, a new set of caching strategies can be generated for the node $\mathbb{L}$ by replacing each child loop in the loop body with brought-up accesses from the child loop and eventually applying the same procedure to the resulting sequence of accesses in the loop body as in Section VI-A. When scanning the allocation and free events of cached patterns to determine the maximum shared memory usage, the required space is increased, upon encountering a child node, by the amount requested by the child node and decreases immediately afterwards.

C. Space Searching and Pruning

By postorder traversing the BST and recording local transformations at each BST node, we guarantee that a node is traversed only when all child nodes have proposed their own caching strategies. The parent node can then explore caching strategies according to all combinations of children's caching strategies. The output is a set of local transformations for the BST subtree rooted at the parent node. In this way, each node is traversed only once, and this dynamic programming method greatly reduces computation overhead and time complexity.

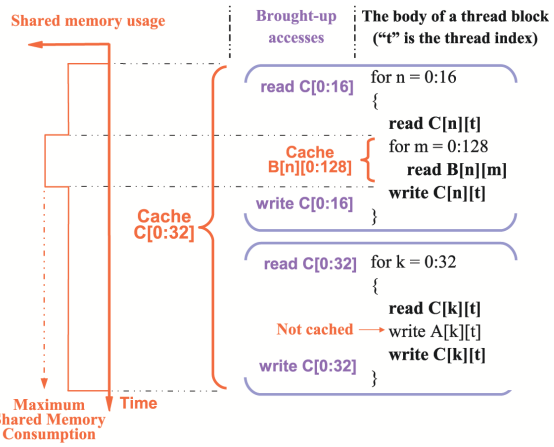


Fig. 3: Generating a caching strategy for an example thread block with two loops and the first loop nested. Assume the caching strategy for the first loop accesses array `C` from shared memory and requests two brought-up accesses for caching `C` in the parent loop (shown in purple). The second loop also inserts two brought-up accesses in the parent loop. These brought-up accesses are merged as one access group and cached altogether. The first loop also loads a slice of array `B` into shared memory every iteration, which is taken into account when emulating shared memory usage.

The search space can easily reach thousands of transformations or more. Using pruning techniques, the search space can be reduced by orders of magnitude. For example, when traversing each BST node, several heuristics can be used to prune the space:

1) Overlapping patterns are either entirely cached or not cached.
2) If two cache strategies for the same BST node have the same global memory accesses but one consumes more cache space, the more space-demanding strategy is discarded.
3) If two cache strategies for the same BST node uses the same cache space but one has additional global memory accesses, the more memory-intensive strategy is discarded.

Our framework also allows users to provide hints for pruning the space. Those hints include what accesses to cache and where to place accesses for shared memory management. Hints are neither critical nor necessary; however, they help reduce the projection time and can sometimes provide additional information for optimization.

The space-searching algorithm extends our prior work [29] by considering *inter-loop* data interactions. Previously investigated transformations, such as loop staging and folding, and pruning techniques, such as identifying beneficial stage sizes, can still be applied.

D. Tradeoff Analyses and Discussions

Tradeoff analyses are performed by comparing the projected performance of all transformations generated from space exploration. For each transformation, we synthesize its characteristics, including the number of active thread blocks, number of computation instructions, and number of memory requests, as well as coalesced and uncoalesced memory instructions. These characteristics serve as inputs to a GPU performance model [18], which mathematically calculates the estimated

performance based on hardware-specific parameters. The performance model also gives insights into whether the workload is compute bound or memory bound. Eventually, the transformation that yields the best projected performance is reported as the "optimized" transformation.

E. Discussion

While our technique has the potential to be integrated into compilers, our goal is not to optimize compilers but to accelerate the system deployment cycle and the application development cycle of "installing $\rightarrow$ coding $\rightarrow$ debugging $\rightarrow$ measuring $\rightarrow$ reinstalling/recoding". Once high-level transformations are suggested by our framework, compilers can then complement the implementation by adding more lower-level optimizations such as instruction scheduling and register allocation. Phase ordering is not an issue in our framework because (1) our space search algorithm follows a strictly ordered procedure, and (2) our technique examines all likely beneficial transformations, rather than making a best guess for a single transformation.

The GPU performance model which we currently employ does not reflect the state-of-the-art GPUs with implicit cache hierarchies [1]; analytical models [37] for these GPUs still require emulation of actual GPU code. Nevertheless, the GPUs studied in this paper are still being actively used in production systems. Our technique about dataflow-driven, multi-loop analysis and transformation is by itself hardware-in; it only utilizes a hardware model whenever a model becomes available. Our multi-kernel GPU performance projection is only to demonstrate the effectiveness of these techniques and the overall framework. We plan to design and incorporate more advanced GPU performance models.

Our technique focuses on data management within and among kernels, which is key to GPU performance. Several transformations are not covered in this paper, including register allocation and instruction scheduling [41]. Transformations such as using atomic operations for global synchronization are not included. We plan to integrate these effects in our future work.

VII. Evaluation

We derive code skeletons from the CPU benchmarks originally written in C++/Fortran. We then apply our techniques to project the optimized performance for individual kernels and fused kernels. The projected performance is then compared with that of the actual GPU implementations which are manually optimized.

Two GPUs, Quadro FX5600 and Tesla C1060, are used in our experiments to validate the projection across GPU generations. Their characteristics are listed in Table II.

TABLE II: Hardware parameters. All other parameters are used by the underlying GPU performance model.

Parameter	FX5600	C1060
Size of the shared memory	16 KB	16 KB
Maximum no. of thread blocks that can run concurrently on one SM	8	8
Maximum no. of warps that can run concurrently on one SM	24	32
No. of threads in a warp	32	32
No. of stream processors	16	30
Clock frequency (GHz)	1.35	1.3
Memory bandwidth (GB/s)	76.8	104.2
Peak flop rate (GFlops)	518.4	933.1
Compute capability (1.3. has better coalescing mechanism)	1.0	1.3

For `HotSpot` and `Stassuij`, we have manually implemented and optimized fused GPU kernels *prior to* developing this technique. For `SRAD` and `CFD`, we implemented the fused GPU kernel according to what our technique suggests, which is reasonably optimized, as described in Section VII-A and Section VII-C. Together, these benchmarks validate how well our technique can transform multiple kernels and how well it can project the benefit of fused GPU kernels.

Performance of the fused kernel is compared with that of separate kernels to show whether fusing kernels may help improve performance and by how much. The comparison is made with both measured data and projected data in order to demonstrate how well our technique can capture the overall performance trend of fused kernels.

We further demonstrate the suggested multi-kernel transformation in the form of code skeletons, which are outputs from the framework as well (Listings 5-8). For illustration purposes, each data access statement is annotated with either "(c)" or "(m)", denoting whether the access touches the shared memory or the global memory, respectively.

All benchmark implementations use single precision. The sizes of input are set according to real input data. GPU implementations are compiled with NVCC over an Intel Xeon E5430 CPU. Host codes are compiled by using GCC 4.0.1 with "-O2" optimization. The GPU kernel execution time is measured by surrounding kernel calls with "cudaThreadSynchronize" and counting the number of CPU cycles in between using the "rdtsc" instruction. Time is then converted to microseconds according to the CPU clock rate. The measured execution times are the average of 10 runs.

The framework itself is executed over a laptop with Intel Core i3 CPU as a single threaded application. The framework is able to include thread block size into the search space; however, when users have a clue about a good thread block size, our technique can take such knowledge as a hint to reduce the search space.

After applying our techniques for multi-kernel analyses and transformations, we synthesize performance characteristics of the transformed code skeleton in the same way as our prior work [29] and use an existing GPU performance model to project execution time [18]. Inaccuracies can stem from several factors, including the following: (1) register usage is not considered, but it can affect the number of active thread blocks per SM; (2) the computation and memory access intensity is assumed to be uniform during execution of a workload; (3) the DRAM memory scheduler schedules memory requests equally for all warps; (4) computation instructions have the same latency; and (5) bank conflicts are not considered. These factors are discussed in more detail in [29] and [18].

A. `SRAD`: Multi-Array Dependency

In `SRAD`, the second kernel (Lines 14-22) in Listing 1 depends on two arrays, `c` and `d`; both are produced by the first kernel (Lines 4-13). With separate kernels, no data is reused, and all data accesses touch global memory. Our technique identifies and analyzes the structure of the fused kernel with a 16×16 thread block size. It first determines that for a $\{\langle 0 : R' : 1\rangle; \langle 0 : C' : 1\rangle\}$ partition of the second kernel, a $\{\langle 0 : R' + 1 : 1\rangle; \langle 0 : C' + 1 : 1\rangle\}$ partition of the first kernel is needed to generate needed elements in both `c` and `d`. Then, 64 caching strategies are explored; the suggested structure of the fused kernel is shown in Listing 5. This fused kernel allocates shared memory space for locally produced `c` and `d` elements and reuses them across codes originally separated between two kernels. Eventually, array `b` is committed to the global memory. We provide the hint in the code skeleton that only array `b` is needed eventually; therefore, the framework does not have to commit the final states of arrays `c` and `d`. This hint is needed

Listing 5: Fused SRAD kernel.

```
R = 4096, C = 4096
float a[R][C], b[R][C]
float c[R][C], d[4][R][C]

grid(R/R', C/C')
thread_block(BlkX, BlkY)
{
  alloc(c) a[-1:R'+2][-1:C'+2]
  alloc(c) d[4][R'+1][C'+1]
  alloc(c) c[R'+1][C'+1]
  load(m) a[-1:R'+2][-1:C'+2]
  for_threads i=0:R'+1,j=0:C'+1
  {
    read(c) a[i-1][j]
    read(c) a[i+1][j]
    read(c) a[i][j-1]
    read(c) a[i][j+1]
    for k=0:4
      write(c) d[k][i][j]
    write(c) c[i][j]
  }
  for_threads i=0:R',j=0:C'
  {
    read(c) c[i+1][j]
    read(c) c[i][j+1]
    read(c) a[i][j]
    for k=0:4
      read(c) d[k][i][j]
    write(m) b[i][j]
  }
}
```

Listing 6: Fused HotSpot kernel.

```
R = 1024, C = 1024, N = 120
float a[R][C], b[R][C], pow[R][C]
/* double buffering */
src=a; dst=b

for n=0:N/2
{
  grid(R/R', C/C')
  thread_block(BlkX, BlkY)
  {
    alloc(c) src[-2:R'+2][-2:C'+2]
    alloc(c) dst[-1:R'+1][-1:R'+1]
    load(m) src[-2:R'+2][-2:C'+2]
    for_threads r=-1:R'+1,c=-1:C'+1
    {
      read(c) src[r][c]
      read(c) src[r][c+1]
      read(c) src[r][c-1]
      read(c) src[r-1][c]
      read(c) src[r+1][c]
      read(c) pow[r][c]
      write(c) dst[r][c]
    }
    swap=src; src=dst; dst=swap
    for_threads r=0:R',c=0:C'
    {
      read(c) src[r][c]
      read(c) src[r][c+1]
      read(c) src[r][c-1]
      read(c) src[r-1][c]
      read(c) src[r+1][c]
      read(c) pow[r][c]
      write(m) dst[r][c]
    }
  }
}
```

Listing 7: Fused CFD kernel.

```
NNB = 4, NVAR = 5
NELR = 193280
float src[NELR*NVAR]
float dst[NELR*NVAR]
float step_factors[NELR]
int neighbors[NELR*NNB]
float fluxes[NELR*NVAR]

grid(NELR/BlkSize)
thread_block(BlkSize)
{
  alloc(c) step_factors[BlkSize]
  alloc(c) fluxes[5][BlkSize]
  for_threads i=0:BlkSize
  {
    for k=1:4
      read(m) src[i+k*NELR]
    write(c) step_factors[i]

    for j=0:NNB
    {
      /* Indirect accesses */
      nb = neighbors[i+j*NELR]
      for k=0:NVAR
        read(m) src[nb+k*NELR]
    }
    for k=0:NNB
      write(c) fluxes[i+k*NELR]

    read step_factors[i]
    for k = 0:NVAR
    {
      read(c) fluxes[i+k*NELR]
      write(m) dst[i+k*NELR]
    }
  }
}
```

for optimization; otherwise the framework assumes the memory states of all arrays have to be preserved as in the CPU code, and all written data will be committed to global memory.

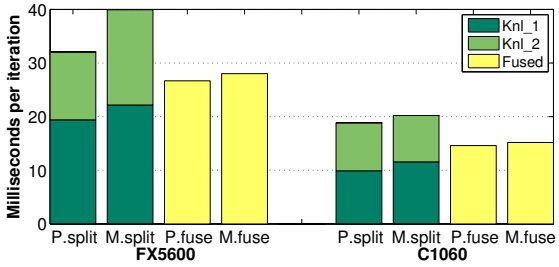


Fig. 4: GPU performance study for SRAD, with both split and fused kernels. M denotes "measured" performance, and P denotes "projected" performance. At the expense of some replicated computation, data reused is increased, and the overall performance is improved by kernel fusion. SRAD has a total of 872M floating point operations, and the fused kernel achieves 31 GFlops and 57 GFlops on FX5600 and C1060, respectively.

Note that in the fused kernel in Listing 5, some elements of array c are computed in multiple thread blocks in order to serve local data dependency within each thread block. This replicated computation does not change the semantics of the code; even if multiple thread blocks attempt to commit the same array element as a result of replicated computation, the resulting committed value would be the same regardless of the order of commitment. In this example, array c is used to hold intermediate data in the fused kernel, and it is not committed to memory.

Figure 4 compares the projected performance with the measured. Because of the improved reuse, we project the fused kernel to reduce run time on both FX5600 and C1060, which is validated by actual measurement. The performance projection error of the fused kernel is 5% for FX5600 and 4% for C1060. The analysis and projection process for separate and fused kernels on both systems are generated within 30 seconds.

B. HotSpot: *Repeated Kernels*

HotSpot repeats over a single kernel in which each iteration consumes a neighborhood of array elements. The original implementation has a global synchronization stage where the kernel is terminated after each invocation, so that in the next invocation the same thread block can safely consume the "halo elements" previously produced by neighboring thread blocks.

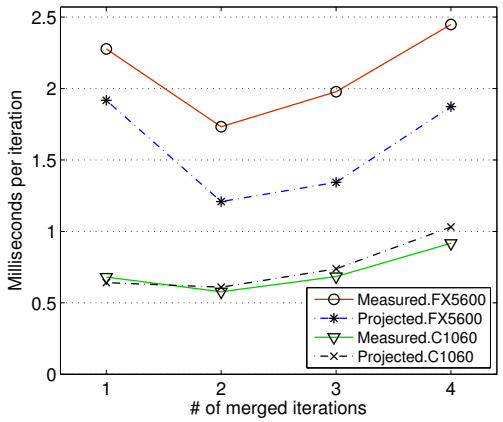


Fig. 5: GPU performance study for HotSpot, with both split and fused kernels. Several iterations of the same kernel can be fused into one. The tradeoff here is the overhead in replicated computation and the benefit of cache reuse. Our technique is able to project the best scheme. HotSpot has a total of 27 M floating point operations per iteration and the best fused kernel achieves 15 GFlops and 45 GFlops on FX5600 and C1060, respectively.

By unrolling the outer loop (Line 6 in Listing 2) and analyzing dataflow between consequent invocations, our framework detects that several invocations of the same kernel can be fused into one. In the fused thread block, a larger partition of a previous invocation is computed in order to serve data for a smaller partition of the following invocation. The resulting

kernel turns out to be an implementation commonly known as "ghost-zone" computation [36]. Listing 6 gives an example of fusing two invocations, but more invocations can be fused. While this technique exploits more cache reuse and reduces synchronization across invocations, the tradeoff is the increase in computation. The more invocations to fuse, the more computation is replicated. Understanding the tradeoffs often requires profiling or actual measurement, and failure to balance the tradeoffs may even degrade performance [28].

Our framework can be used to balance such tradeoffs without special care for this case. We instruct our framework to fuse up to four invocations. We also specify a thread block size of 16×16 and provide the hint that only `dst` is the necessary output for the fused kernel. Up to 16 transformations are explored in each fused kernel. As Figure 5 shows, the actual implementation determines that fusing two invocations yields the best performance, which aligns with our projections. The performance projection error is 16-30% on `FX5600` and 5-12% on `C1060`. Projections are generated in 40 seconds.

C. `CFD`: *Multiple Dependent Kernels*

In `CFD`, the third kernel (Lines 33-41) in Listing 3 depends on the first kernel (Lines 9-17) for array elements in `step_factors` and the second kernel (Lines 33-40) for array elements in `fluxes`. Our technique attempts to fuse all three kernels into one. It turns out that the loop iterators for all three kernels have a direct one-to-one correspondence and their loop bodies can be concatenated. Then 256 caching strategies are examined. Elements in `step_factors` and `fluxes` are reused across kernels and are therefore allocated in shared memory; other elements need not be cached.

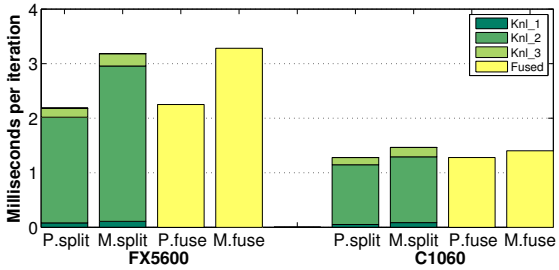


Fig. 6: GPU performance study for `CFD`, with both split and fused kernels. `M` denotes "measured" performance, and `P` denotes "projected" performance. Our projection determines that fusing kernels is not beneficial due to the dominance of un-coalesced indirect accesses. `CFD` has a total of 73M floating point operations, and the fused kernel achieves 22 GFlops and 52 GFlops on FX5600 and C1060, respectively.

The resulting fused structure is shown in Listing 7. In this fused kernel, `src` may be reused or shared among threads at Line 23, depending on the value of the indirect index. In order to guarantee such cache reuse, however, the entire array has to be brought into shared memory, a strategy that is infeasible. Therefore, this access is not cached. In addition, elements in `fluxes` are produced in an inner loop (Line 26), and our framework is able to detect the possibility of reusing it at another inner loop (Line 31). In this example, a thread block size of 128 is used. We provide a hint that `dst` is the only needed result and that other arrays need not be committed into global memory.

Figure 6 compares the projected performance with the measured. Because of register pressure, fewer warps can run simultaneously to hide latency; such an effect is not considered in the GPU performance model, and therefore the projected performance is noticeably lower than the measure. Nevertheless, the overall trend is still captured. Although the total number of global memory accesses is reduced in the fused kernel to some degree, the major overhead stems from the un-coalesced indirect accesses at Line 23, which remains in the fused kernel. As a result, our framework indicates little benefit in fusing the kernels, and this phenomenon is observed from the measurements. The performance projection error of the fused kernel is 31% for `FX5600` and 9% for `C1060`. The process to obtain all projections in Figure 6 takes 6 minutes.

D. `Stassuij`: *Different Loop Shapes with Indirect Indices*

`Stassuij` has two dependent kernels with different loop spaces. Moreover, the inter-kernel data dependency caused by array `B` involves indirect indices in array `M`. Nevertheless, our framework is able to fuse the two kernels into one kernel in the form of Listing 8. To assess the degree of coalescing, we profiled the average data stride of indirect indices in array `M` and provide this hint to the framework. The loop index `i` of the first kernel (corresponding to Line 11 in Listing 4) is now replaced by an expression involving loop indices of the second kernel (Line 17 in Listing 8) and indirect accesses to array `M`.

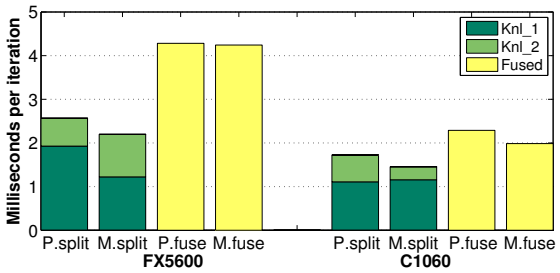


Fig. 7: GPU performance study for `Stassuij`, with both split and fused kernels. `M` denotes "measured" performance, and `P` denotes "projected" performance. Our projection indicates that the fused kernel has worse performance due to the increase in un-coalesced memory accesses. `Stassuij` has a total of 18M floating point operations, and the fused kernel achieves 4 GFlops and 9 GFlops on FX5600 and C1060, respectively.

Listing 8: Fused `Stassuij` kernel.

```
1  NT = 132, NS = 2048, NSG = 512
   ELEMS = 1848, avg_j_ntdt = 14
3  int J[NT+1], I[ELEMS], T[ELEMS]
   float A[NT][NS][2], B[NT][NS][2]
5  float C[NT][NS][2]

7  grid(1, NSG/NSG', 1, 1)
   thread_block(1, NSG', 4, 2)
9  {
     alloc(c) M[0:NSG'][0:4]
11   load(m) M[0:NSG'][0:4]
     alloc(c) J[0:NT+1]
13   load(m) J[0:NT+1]
     for_threads g=0:NSG', m=0:4, r=0:2
15   {
       i = M[g][m]
17     StageSize = 11
       alloc(c) T[0:avg_j_ntdt*StageSize]
19     alloc(c) I[0:avg_j_ntdt*StageSize]
       // Each thread processes NT iterations
21     for j = 0:NT
       {
23       alloc(c) B[0][NSG'*4][2]
         read(c) J[j]
25       read(c) J[j+1]
         for n = 0:avg_j_ntdt
27       {
           read(c) T[n]
29         read(c) I[n]
           read(m) A[I[n]][i][r]
31       }
         write(c) B[j][i][r]
33
         alloc(c) A[0][NSG'*4][2]
35       load(m) A[0][NSG'*4][2]
         read(c) A[j][M[g][m]][r]
37       read(c) B[j][M[g][m]][r]
         write(m) C[j][M[g][m]][r]
39     }
     }
41 }
```

The benefit of fusing the two kernels in `Stassuij` is not obvious to program developers. Although elements in array `B` can now be reused in shared memory, it comes with a cost; since loop index `i` in the first kernel now becomes a nonlinear expression in the fused kernel, accesses to array `A` (Line 30 in Listing 8) are no longer coalesced. Conventionally, such a tradeoff can be analyzed only after the actual code is implemented and tested.

Because different values of `j` correspond to rows in the sparse matrix with variable lengths, partitioning the loop space along the `j` dimension leaves the per-thread workload unknown and it may lead to significant load imbalance. This high level knowledge is passed to our framework, which determines that instead of partitioning the loop space along `j` for parallel threads, each thread loops over `j` internally.

After examining 128 transformations for the fused kernel, our framework projects that the un-coalesced accesses cancels the benefit of reduced global memory accesses, and kernel fusion is not beneficial in this case. Figure 7 compares the projection with the actual implementation. Both systems show similar trends, and the projection is validated. The performance projection error of the fused kernel is 1% for `FX5600` and 15% for `C1060`. All projections are made within 2 minutes.

E. Sources of Inaccuracy

The sources of inaccuracy can stem from both the synthesized characteristics and the underlying GPU model. For example, the effects of register pressure, bank conflicts, partition camping [2], and long-latency instructions are not considered. Other factors that contribute to inaccuracy include: (a) the average data stride may not linearly correspond to coalescing effectiveness, especially for C1060, where the coalescing mechanism is more dynamic (e.g. Stassuij); (b) the GPU model assumes computation instructions and memory accesses are uniformly distributed during execution.

VIII. Related Work

Current GPU programming tools including compilers and code translators produce GPU code or annotated binary legacy code [43], [6], [27], [12]. GPU performance tools [23], [40] have also been developed. All these tools rely on developers to specify well-defined parallel `for` loops as individual kernels. Many of these techniques are domain-specific, and they often require physical GPUs to evaluate the performance of optimized code.

A few other techniques attempt to optimize data communication between CPU and GPU, given user-created GPU kernels [32], [3], [19]. These techniques do not define and build GPU kernels that optimizes data movement *within* the GPU. Our framework is complementary to these techniques.

Recently, GPU performance models have also been proposed to help understand the runtime behavior of existing codes [4], [18], [24], [45], [37]. Although these tools provide in-depth performance insight, they do not explore various code transformation to overcome performance bottlenecks. Some of these techniques also require dynamic profiling on a physical hardware [31] or an emulator [11].

Several cross-platform projection and tuning techniques study performance on inaccessible systems [26], [38], [44], [25], [39], [8]. Often, the relationship between tunable parameters and application performance is gleaned according to profiled metrics or simulated data. However, performance prediction tools such as PBound [30] estimate performance bounds, namely, the projected performance if the application is computation-bound memory-bound. However, these techniques are not able to explore transformations in code structures and caching strategies, and it is often difficult to identify performance bottlenecks with these tools.

The Thrust library [7] relies on developers to specify which GPU kernels to fuse; functions to be fused typically have element-wise dependency or exhibit a reduction pattern. Its ability to explore caching strategies is also limited. Fousek et al. studied automatic GPU kernel fusion [13]. Without comprehensive dataflow analysis, their technique preserves the way each kernel is partitioned into thread blocks, and merges two thread blocks with two assumptions: (1) inter-kernel dependency is restricted within each pair of producer and consumer thread blocks and (2) all data resides in shared memory. These assumptions do not apply to the generic case. Moreover, having all data in shared memory may be suboptimal because it can limit the number of warps to hide latency; it may also be inapplicable because of shared memory's limited capacity.

Loop fusion has been widely researched in the context of CPU and embedded systems [14], [35]. The state of the art uses polyhedral models to transform multiple loops into fewer loops and schedule individual instructions. However, such models are constrained to loops with iterators that have direct correspondences, and their ability in dataflow analysis and cache optimization for different architectures is limited.

Domain-specific performance modeling techniques have been proposed for different hardware systems [10], [28], [15], [22], [17], [5], [21]. Our transformation framework is not limited to a particular application domain or a particular hardware.

IX. Conclusion and Future Work

We have proposed a technique to transform and fuse a sequence of parallel operations on GPUs and project their overall performance. Our technique can identify dependent regions between producer and consumer kernels, derive a shared loop space, and optimize dataflow in the fused kernel. The key challenges addressed include inter-loop iteration dependency analysis, hierarchical dataflow analysis and caching strategy exploration for multiple, nested loops.

With four distinct multi-kernel benchmarks, we demonstrate that performance benefit of kernel fusion may vary in different cases because of tradeoff factors including computation intensity, memory accesses, and data access strides. With our technique, we are able to project the GPU performance for fused kernels *without* needing to implement GPU code or accessing the physical GPU hardware. Projections are compared with measured data on two generations of GPUs. The projection error for our experiments averages at 14% and a maximum of 31%. Our technique is able to predict situations where the performance effects of fusing multiple kernels can be beneficial, harmful, negligible, or vary based on the degree of fusion. The work has been integrated into the GROPHECY framework [29], and will be available online at perfengr.alcf.anl.gov/grophecy.

Acknowledgments This work was supported by the U.S. Department of Energy under contract DE-AC02-06CH11357. The research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We thank Christopher Wallis for creating the web interface for GROPHECY. We also thank Gail Pieper for her careful editing of the manuscript.

REFERENCES

[1] NVIDIA's next generation CUDA compute architecture: Fermi. *NVIDIA Corporation*, 2009.

[2] A. M. Aji, M. Daga, and W. Feng. Bounding the effect of partition camping in GPU kernels. In *CF*, 2011.

[3] C. Augonnet, S. Thibault, R. Namyst, and P-A Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23:187–198, February 2011.

[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, 2010.

[5] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and José Carlos Sancho. Using performance modeling to design large-scale systems. *Computer*, 42, November 2009.

[6] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *CC*, 2010.

[7] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. 2011.

[8] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *ICS*, 2011.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphisc processors using CUDA. *JPDC*, 2008.

[10] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP*, 2010.

[11] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Proceedings of The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010.

[12] D.Unat, X.Cai, and S. Baden. Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C. In *ICS*, 2011.

[13] Jan Fousek, Jiři Filipovič, and Matuš Madzin. Automatic fusions of CUDA-GPU kernels for parallel map. In *HEART*, 2011.

[14] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *ISSS*, 2001.

[15] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *ICS*, 2011.

[16] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Parallel Distrib. Syst.*, 2, 1991.

[17] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *SC*, 2011.

[18] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[19] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. PLDI, 2011.

[20] M. H. Kalos, M. A. Lee, P. A. Whitlock, and G. V. Chester. Modern potentials and the properties of condensed ^{4}He. In *Phys. Rev. C 66, 044310-1:14*, 1981.

[21] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC*, page 37, 2001.

[22] Darren J. Kerbyson and Adolfy Hoisie. Performance modeling of the blue gene architecture. In *John Vincent Atanasoff Symposium*, pages 252–259, 2006.

[23] A. Klckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. *CoRR*, 2009.

[24] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU platform. In *HiPC*, 2009.

[25] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII*, 2006.

[26] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP*, 2007.

[27] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC*, 2010.

[28] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *ICS*, 2009.

[29] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *SC*, 2011.

[30] Sri Hari Krishna Narayanan, Boyana Norris, and Paul D. Hovland. Generating performance bounds from source code. In *Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, 9 2010.

[31] NVIDIA Corporation. NVIDIA Corporation. NVIDIA Visual Proler. http://developer.nvidia.com/content/nvidiavisualprofiler.

[32] J. A. Pienaar, A. Raghunathan, and S. Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *ICS*, 2011.

[33] S. C. Pieper, K. Varga, and R. B. Wiringa. Quantum Monte Carlo calculations of A=9,10 nuclei. In *Phys. Rev. C 66, 044310-1:14*, 2002.

[34] S. C. Pieper and R. B. Wiringa. Quantum Monte Carlo calculations of light nuclei. In *Annu. Rev. Nucl. Part. Sci. 51, 53*, 2001.

[35] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL*, 2011.

[36] G. Rivera and C.-W Tseng. Tiling optimizations for 3D scientific computations. In *SC*, 2000.

[37] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. GPUPerf: A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*, 2012.

[38] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC*, 2002.

[39] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, March 2003.

[40] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. m. W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *LCPC*, 2008.

[41] V. Volkov. Better performance at lower occupancy. Presentation at NVIDIA GTC, 2010.

[42] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. GREENCOM-CPSCOM, 2010.

[43] M. Wolfe. Implementing the PGI accelerator model. In *GPGPU*, 2010.

[44] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC*, 2005.

[45] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.